PERFORMANCE STUDY OF GRAPH CONVOLUTIONAL NETWORKS FOR MEDICAL PREDICTION-BASED NETWORKS

An Undergraduate Research Thesis

by

BEN D'ANTONIO

Submitted to the Computer Science Honors program at
Texas A&M University
in partial fulfillment of the requirements for the designation as a

COMPUTER SCIENCE HONORS UNDERGRADUATE

Approved by Research Advisor: Dr. Anxiao Jiang

May 2019

Major: Computer Science and Engineering

TABLE OF CONTENTS

ABSTRACT

Performance Study of Graph Convolutional Networks for Medical Prediction-Based Networks

Ben D'Antonio
Department of Computer Science and Engineering
Texas A&M University

Research Advisor: Dr. Anxiao Jiang
Department of Computer Science and Engineering
Texas A&M University

Predicting the effects of Polypharmacy is a difficult task, and a great amount of money is spent annually remedying the effects of negative drug interactions arising from Polypharmacy. However, Machine Learning can be used to give more accurate predictions than traditional means. In this thesis, we survey current methods of applying Machine Learning to Polypharmacy. We rigorously define a theoretical Polypharmacy problem and design a Graph Convolutional Network that can learn to strongly model our problem. We discuss its performance and offer future steps for generalizing the model to gain a better understanding of the field of Polypharmacy and the potential of Machine Learning to improve it.

NOMENCLATURE

AI	Artificial Intelligence
DL	Deep Learning
DNN	Deep Neural Network
GCN	Graph Convolutional Network
ML	Machine Learning

LIST OF FIGURES

FIGURE Page

1. INTRODUCTION

Polypharmacy is the simultaneous use of multiple drugs to treat a single disease or condition. Typically, the concurrent use of five or more drugs is considered Polypharmacy [1]. The process of developing a single drug approved for human-use is arduous because of the complex interactions any drug will have with the human body. The cost of taking a drug from concept to market is estimated at $1.3 billion with an approval rate of around 1 out of 1000 [2]. When multiple drugs are used simultaneously to treat a disease, there is a significant chance that a side-effect will arise from the drugs' interactions, and the result can greatly harm a person's health. Each year, the U.S. alone spends an estimated $30 billion to $180 billion addressing issues caused by Polypharmacy [3]. These side-effects are often harder to identify than side-effects arising from a single drug [4]. This problem is further exacerbated by resource limitations, as rigorous testing of a drug's interaction with all marketed drugs is infeasible. Therefore, there is much to be done to improve Polypharmacy practices.

One method to progress Polypharmacy's state of the art is through the application of Machine Learning (ML). ML is the process of having Artificial Intelligence (AI) learn to perform a task without being provided explicit instructions. Instead, ML processes use large bodies of data to identify patterns and make inferences based on them. For example, given New York's weather data over the past twenty years, a ML process could learn to predict what New York's weather will be tomorrow. Because the ML process is provided no explicit instructions, however, it would have to identify the best metrics to predict the weather, such as the weather the day before, on its own. In the beginning, like a child, the ML process would do no better than naively guessing. To become more intelligent, the ML process must practice like a human would. That is, the ML process needs to make

a prediction, see if it was right or wrong, learn from it, and give a better prediction next time. However, predicting the weather once a day is not the fastest way to learn. Instead of telling the ML process to predict New York's weather for tomorrow and then waiting a day to see if its prediction was correct, the ML process can pick a date in the past, hide the actual answer from itself, and then try to predict that date's weather based on the weather data before it. After practicing enough, through trial and error, it would learn to make its prediction based on good metrics like the weather that day in past years, the weather leading up to that day, and historical trends in New York's weather. Through this cycle of training, testing, and learning from feedback over a large body of accurate data, a ML process can quickly develop expertise over a specific problem and make predictions like a human would.

Because there exist large bodies of historical Polypharmacy data–and each drug's properties are typically well understood–Polypharmacy stands to benefit greatly from ML. Namely, using historical data about drugs; their interaction with specific diseases, disorders, and proteins in human bodies; and any interactions with other drugs; ML can be trained to predict the complex interactions that occur in Polypharmacy. Some of the tasks ML can assist Polypharmacy in are the following:

- Identifying problematic interactions between drugs.
- Reinforcing previous findings in Polypharmacy.
- Providing a quick-access method of determining the risks of a set of drugs.
- Finding new, beneficial drug interactions in Polypharmacy.

The results of these benefits are the following:

- Reducing associated costs of Polypharmacy, including mortality rate and economic resources.

- Increasing our knowledge and understanding over the field of Polypharmacy.
- Centralizing and organizing Polypharmacy's massive body of knowledge.

Thus, the motivation of this research is to assist in improving Polypharmacy with ML. In this thesis, we survey current methods of using ML to improve Polypharmacy, propose a theoretical model for a Polypharmacy data set, and create a Graph Convolutional Network (GCN) model to train on the data set. The data is generated from a low-dimensional (latent) space but is observable only in a high-dimensional (observation) space. This means that the data is actually dependent on a small number of factors but is artificially stretched to seem dependent on many factors. The GCN sees only the observation space of the data and learns to make accurate predictions on the data set. The data set, while simple, is nontrivial. We detail and discuss the knowledge gained from simulating a theoretical, simple but nontrivial Polypharmacy data set on a GCN.

1.1 Current Methods of ML for Polypharmacy

The predominant subarea of ML being applied to Polypharmacy is Deep Learning (DL). DL is a subset of ML inspired by how humans think and learn. DL models are generally based on the structure and function of the brain itself. DL is applied to image classification, outcome prediction, Natural Language Processing (NLP), and many other actions that humans perform on a day-to-day basis. Deep Neural Networks (DNNs) is a subarea of DL that epitomizes the concept of mimicking the human brain as a ML network. DNNs are currently a popular ML model used in the field of Polypharmacy. As we will see, DNNs have properties that are very useful for Polypharmacy.

In the human brain there are many interconnected cells called neurons. The relationship between these neurons can be thought of as a very complex directed graph where each node is a neuron. A neuron N receives impulses from neurons that have a directed edge to N; and, depending on the strength of the impulses, N may activate and send impulses

to the neurons it has directed edges to, which may then activate other neurons, causing a cascade effect. The graph of neurons has many source and sink neurons too. For example, if an object collides with a person's leg just below the knee, it will cause a physical sensation that triggers a source neuron to send impulses to its connected neurons. Eventually, the impulses will cascade far enough that the impulses reach the neurons responsible for controlling the thigh muscle. The impulses will cause the leg muscle to contract, causing the person's knee to automatically kick out. This is how human reflexes work. Depending on how the graph of neurons, the neural network, is set up, certain stimulations can deterministically cause certain reactions, such as reflexes.

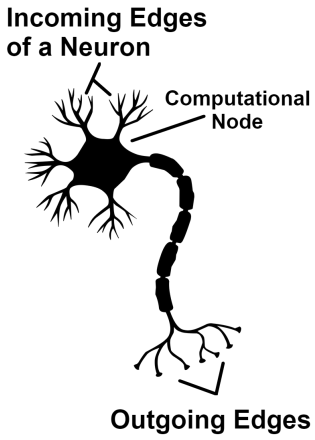

Figure 1.1: A neuron receives electrical signals from other neurons. The neuron will send a signal to its outgoing neurons depending on the strength of each impulse and how the neuron values each incoming signal.

This neural network model also has the capability to learn from trial and error. To do so, a neuron can value the impulses of its incoming connections independently; one incoming neuron's impulse may be valued as half as significant as another neuron. As an action is performed repeatedly, based on feedback, a neuron may respond less and less to

a certain neuron's impulse but more so to another neuron's. For example, when a person attempts a task and does it right, the person feels happy. This feeling of happiness can be feedback to the neural network; thus, the neurons involved with the task will start to value the incoming impulses that resulted in positive feedback more. When the person does the task wrong, the inverse will occur. Eventually, the neurons will determine which incoming impulses generate the best result. This is how muscle memory is developed. There is certainly much more to the field of Neuroscience, but this is sufficient for explaining the fundamentals of DNNs.

Because of a neural network's abilities to deterministically compute outputs based on certain inputs and to learn from trial and error, it is unsurprisingly a powerful computational model which enables human intelligence. Fortunately, it is also a computational model that can be approximately captured by computers.

1.2 Deep Neural Networks

As mentioned, a Neural Network can be modeled in Computer Science as a directed graph consisting of thousands, millions, or more densely connected nodes, where each is a neuron. These nodes are organized into layers with a unidirectional data flow. There can be source and sink nodes throughout the network; however, generally, there is a single input layer at the start of the network and a single output layer at the end. The layers in between are called hidden layers, and they perform the computations. When an input is received, depending on the intensity, the corresponding source node will activate with an intensity measured in the form of a number, typically ranging from 0 to 1. The source node then sends this number to each of its outgoing connections. For a non-source node N, each of N's incoming connections are assigned a "weight"; some incoming connections are significantly more important than others and, thus, have a higher weight. The node takes the number sent through the incoming connection–the "strength" of the incom-

ing connection's activation–and multiplies it by the weight assigned to that connection. If there are multiple activated incoming connections, each of their values are weighted and then summed. The node will then scale this number to the range 0 to 1, where 1 means the activation is as strong as possible, and send this number to each of its outgoing connections. This propagates throughout the network and eventually arrives at sink nodes, where output is generated. For a Neural Network, there can be many inputs. Returning to the previous example for ML, one input to the Neural Network may be whether it rained the day before; another input may be the average temperature for the previous week; and so on. The outputs may be whether it will rain today and what the average temperature will be.

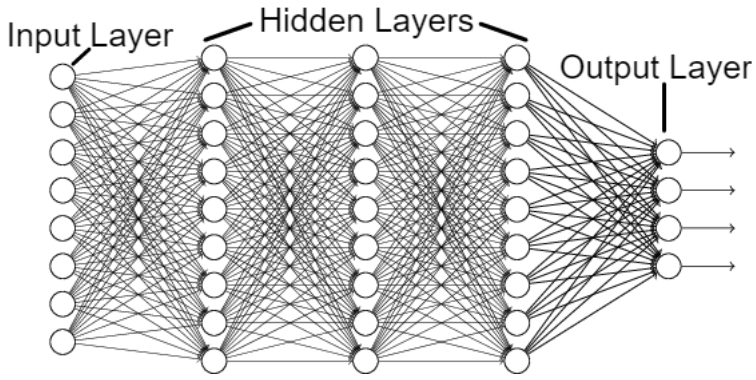

Figure 1.2: Structure of a DNN.

A Neural Network is governed by weights, represented as numbers, which are determined based on feedback. However, in the beginning, a Neural Network has had no feedback and, thus, knows nothing. Unlike human brains, which have had millions of years of feedback and start off with a well-tuned network, a Neural Network is created at a moment's notice and must learn to solve a complex problem in a short time span. To address this, all nodes in the Neural Network initially receive random connection weights.

Therefore, the network will do no better than randomly guess at the start. However, this initialization scheme allows the network to quickly identify how helpful each node is early on from its feedback. For example, if a node has some incoming connection weight initialized as .999 but consistently gets the answer wrong, it may lower this weight and increase the weight of its other incoming connections. Eventually, the network will instead begin to consistently give correct outputs and proceed to reinforce what it has learned so far.

A DNN is simply a Neural Network with multiple hidden layers. More layers in a DNN allow for the potential to model more complex problems; however, too many layers can cause a network to become slow to learn because it has too factors to consider when tuning itself. Such a trade off is a common balancing act in ML. For example, this problem also affects the choice of the number of nodes in each hidden layer.

ML learning is divided into two alternating phases: training and testing. Both phases provide a set of inputs to the DNN and evaluate the network's performance. During training, the network tunes its values based on feedback from its prediction attempts; this is where learning takes place. Unlike training, testing does not provide feedback to the network. Instead, testing is used to simply evaluate the network's performance. To evaluate a network's performance properly, it is important that the testing is unbiased. To accomplish unbiased testing, the training and testing data sets are kept mutually exclusive. Thus, the network will never see testing data during its training phase. The network will, therefore, never receive feedback from any data in the testing set. Because of this, the testing results are unbiased and serve as a good indication of the network's actual grasp of the problem it is solving. This presents another balancing act, however: because a network only receives feedback from its training phase, it may simply learn to give the correct answer for input that is similar to its training data. Thus, when it finds unfamiliar data for the same problem, it will predict incorrectly. This is called overfitting. Overfitting is indicated by a high training accuracy but a low testing accuracy when making predictions. On the other

hand, the network may be learning the general problem but is not doing particularly well overall. This is called underfitting. Underfitting is indicated by both a low training and low testing accuracy when making predictions. It is important to note that having a low training accuracy and a high testing accuracy is very unlikely to occur because the network exclusively fits itself on the training data. There are many parameters of DNNs that can cause overfitting and underfitting; to name a few: the number of hidden layers, the number of nodes in each hidden layer, the activation functions of each node, and the rate at which random dropout of learned information occurs (to combat overfitting).

A GCN is a specific type of DNN. What has been covered thus far is enough to introduce GCNs.

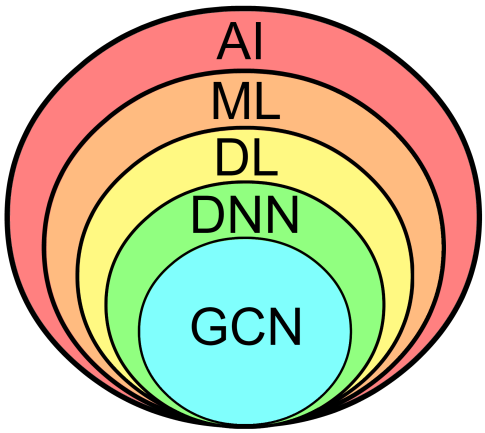

Figure 1.3: The relationship between the discussed forms of AI, from most general to most specific.

2. RELATED WORKS

As first proposed by T. Kipf and M. Welling, a Graph Convolutional Network (GCN) notably is a DNN that learns problems that can be modeled by graphs [5]. Many problems that Computer Science is concerned with can be modeled by graphs, including social networks, the World Wide Web, and, of particular interest to our research, protein-interactions in Polypharmacy [6][7][8]. For example, in a social network, each person can be represented as a node in a graph. If two people are mutual friends, an undirected edge exists between them. Given an incomplete graph of a social network, a GCN can learn to predict whether two people will be friends with each other and, thus, complete the graph. It may learn to do so by observing trends in the graph. For example, if a person A has many mutual friends with a person B, person A and person B are statistically likely to also be friends. Additionally, a GCN can be supported by external information about each node, called features. For example, in a social network, if everyone in the network takes a personality test, then a GCN may observe that people with very similar personalities tend to more often be mutual friends. Combining the observable edges between nodes with feature information about each node, a GCN can learn to effectively predict things in a graph like missing edges. Because Polypharmacy datasets are typically large, complex graphs, GCNs are an excellent candidate for modeling these problems.

Recently, a lot of research has been done on the prospect of applying ML to Polypharmacy; this research has focused on laying the groundwork for applying ML techniques like GCNs to Polypharmacy [9] [10] [11]. Overall, GCNs have sparked great interest as a research topic, which focuses on improving GCNs' scability and power and creating new ML models based off GCNs [12] [13] [14]. Some of this research, such as creating an attention-based GCN, may be applicable to this research in the future.

2.1 Graph Convolutional Networks

A GCN takes as input a data representation of a graph (commonly an adjacency matrix) and a feature matrix. A "feature" is a property of each node in the graph and is represented as a number. For example, a feature in a group of people forming a social network may be a measure of how patient each person is. The feature matrix is 2-dimensional. Each row in the feature matrix is a feature vector corresponding to one node in the graph, a measure of each node's features. Each column in the feature matrix corresponds to a certain feature which we believe is useful to our network for understanding the nodes and graph. Thus, given a row and column, the location in the feature matrix is a feature value that represents a specific node's relationship with a specific property we believe useful to the network in the form of a number. For example, in the Myers-Briggs Type Indicator test, there are four personality traits (features) studied:

- Extraversion vs. Introversion
- Sensing vs. Intuition
- Thinking vs. Feeling
- Judging vs. Perceiving

We can represent each of these four features on a dense scale of 0 to 1, where a 0 means that the person strongly exhibits the left trait, a 1 means the person strongly exhibits the right trait, and a $.5$ means the person exhibits both traits to a moderate degree. If we suppose we have a social network consisting of four people, then, after the personality test, each of the four people will have four feature values, comprising four feature vectors in the feature matrix:

$$\begin{pmatrix} 1 & 0 & .5 & .75 \\ .6 & .9 & .72 & .54 \\ 0 & .63 & .5 & 0 \\ .8 & .6 & .1 & 1 \end{pmatrix}$$

The first row is person A's feature vector. It indicates that person A is strongly Introverted, strongly Sensing, moderately Feeling and Thinking, and fairly Perceiving.

Then, let us suppose that following is the adjacency matrix describing these four people's social network:

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

Now we have established the inputs which the GCN uses to learn. The GCN learns from these inputs through a process called convolving, which means entwining together. The network entwines/combines/convolves feature information about each node together with the provided graph in an attempt to understand why certain edges exist. We will describe how a GCN uses convolving to learn.

A GCN, as with any DNN, consists of a number of hidden layers surrounded by an input and output layer. A GCN's string of hidden layers begins with a series of graph-convolutional layers. Each graph-convolutional layer takes as input the adjacency matrix representation of the graph, A, and a feature matrix, H, and outputs another feature matrix with an arbitrary number of features. If the input feature matrix is $(N \times F)$, the output feature matrix is $(N \times F')$, where F' is an arbitrary number (typically smaller than F). The output feature matrix is the convolved information of A and H. This means that the

output feature matrix contains the information of the input feature matrix combined in some way with the information contained in the adjacency matrix. There are many ways to "combine" the two matrices; one such method will be discussed briefly. Because we can arbitrarily choose the number of output features of a graph-convolutional layer, we can set up a series of graph-convolutional layers that take A and H_{i-1} as input and output H_i:

$$H_i = f(A, H_{i-1}).$$

There are many ways to combine A and H_{i-1} to obtain H_i; we provide following simple example:

$$H_i = \sigma(AH_{i-1}W_i)$$

In the above function, σ is some activation function that operates on the product $AH_{i-1}W_i$. It is commonly a ReLU function. A is the adjacency matrix; H_{i-1} is the feature matrix from the previous layer; and W_i is the weight matrix of the current graph-convolutional layer. A is $(N \times N)$, where N is the number of nodes in the graph; H_{i-1} is $(N \times F_{i-1})$, where F_{i-1} is the number of features from the previous layer; and W_i is $(F_{i-1} \times F_i)$, where F_i is the number of features of the current layer. Thus, when multiplied in series, the output matrix H_i is $(N \times F_i)$, which is a feature matrix with an arbitrary number of features.

Multiplying the adjacency matrix with the feature information serves as a way to blindly convolve/combine the two together. Further weighting different values of the resulting matrix with the network's own weight matrix allows the network to learn which features-edge relationships are most relevant to accurately making predictions. In practice, $f(A, H_{i-1})$ is typically more complex than our example, but this is sufficient to have a general understanding of the process.

After passing in the initial feature matrix and adjacency graph into the first layer, successively passing the output (and the adjacency matrix) through more graph-convolutional layers while downsizing the number of resulting features of each output feature matrix allows a GCN to obtain a wholistic view of the provided information's trends, from more specific in earlier layers to more general in later layers. After obtaining the output of the final graph-convolutional layer, we have a compacted, generalized feature matrix of the original input that hopefully contains as much relevant information as possible. We can pass this feature matrix as input into standard DNN layers to make predictions about the graph.

As a general example of a GCN problem, let us suppose we have an incomplete graph that contains all nodes but not all edges. We then want to predict the missing edges into the graph. In practice, providing the GCN with the entire adjacency matrix would not be very helpful, especially if the graph were very large. It would be very difficult for the network to make extrapolations from the entire data set. Thus, instead of providing the entire adjacency matrix and feature matrix, we can provide subparts of the graph and feature matrix to the GCN. There are many ways to do so. For example, we could select a random edge from the graph and provide the two feature vectors of the nodes forming the edge. The network would then try to learn, from one edge at a time, why two nodes share an edge. However, this would be very slow. A better approach would be to provide the network with a subgraph and a subfeature matrix. This subgraph can be constructed by randomly selecting a few nodes and forming an adjacency matrix between them. The subfeature matrix would then simply be the matrix consisting of each of the selected nodes' feature vector. This allows the dataset to be small enough for the network to learn significant information while not providing so little of the graph that it can not see general trends. Additionally, this combats overfitting, as showing the entire graph to the network repeatedly would bias the network into just "memorizing" the whole graph. Thus, it would be

less likely to predict the missing edges in because it has repeatedly observed that the two nodes do not share an edge during training.

2.2 Decagon

Of particular interest to this research, Decagon is a GCN created to train on a real-world Polypharmacy data set [8]. Decagon is designed with the goal of predicting the safety of drug combinations in Polypharmacy. Decagon's data set consists of several databases of drug and protein information aggregated into a massive graph problem. The data includes many cross-interactions of these drugs and proteins. For example, if two proteins found in the human body are known to interact with each other, then in Decagon's graph, the proteins share an edge. Likewise, if a drug is known to interact with a protein, they share an edge. These edges are undirected and are simple; there is only one type of edge that can exist between protein-protein and drug-protein pairs which indicates that the two interact. Similarly, if two drugs are known to have an interaction, called a side-effect, then an edge exists between the two drugs. In contrast, however, this edge contains information about the specific side-effect (and is, thus, not simple). For example, if two drugs' interaction results in gastrointestinal bleeding, then the edge shared between them is a gastrointestinal bleeding edge. If two drugs' interactions result in multiple side-effects, then they share multiple edges. The end result is a massive, complex graph with two node types (drugs and proteins) and many edge types (protein-protein edges, drug-protein edges, and a large set of possible drug-drug side-effect edges). The following is Decagon's visualization of their graph:

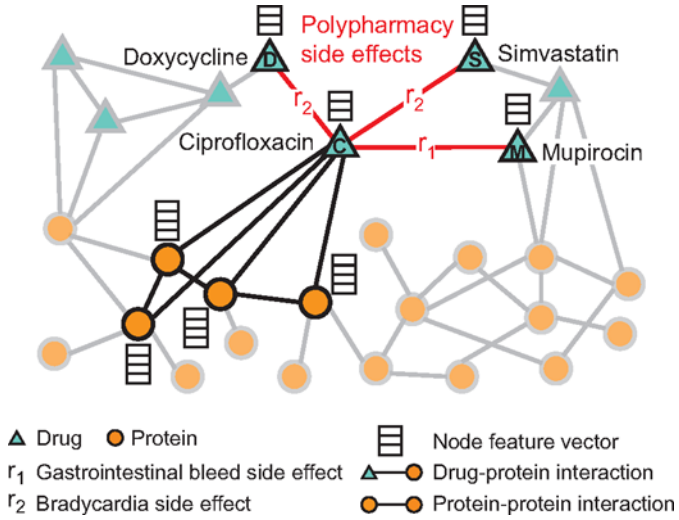

Figure 2.1: Decagon's Visualization of its Graph.

Decagon also has feature information about each of the proteins and drugs which are based on observed, general properties of the drug-protein data set. This is certainly a very complicated Polypharmacy problem, which is further complicated by the uncertainty of its data being observed and not guaranteed to be correct. Additionally, some of the selected features may not be relevant to determining interactions.

Decagon tests its network by removing some edges from the graph and attempting to repredict them back in, similar to the previous example of a GCN problem. Despite the complexity and uncertainty of the data, Decagon appears to be fairly successful. However, there is much room for improvement. In particular, removing complexity from the problem and then experimenting on the resulting subproblem may allow for a better understanding of the problem as a whole.

3. GCN MODEL

To remove complexity from Decagon's problem, we propose the following simpler but nontrivial, theoretical Polypharmacy graph for which we can design a GCN model and learn to make predictions on:

- Generate a drug graph where an edge between two drugs indicates that they cause a side-effect.
- There is only one node type and one edge type. Thus, the graph is simple and undirected.

The following is an example drug graph:

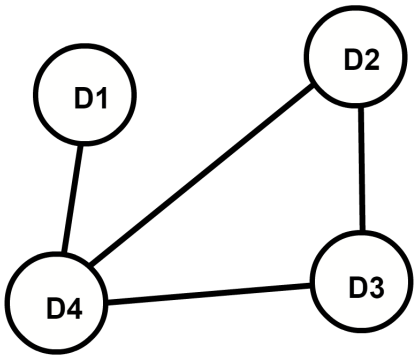

Figure 3.1: An example drug graph.

If the graph were entirely randomly generated, there would be nothing to learn. Therefore, we propose the following method of generating the drug graph:

- Generate N random (x, y) pairs, where N is the number of drugs in the graph and $0 \leq x, y \leq 1$. Each (x, y) pair is a drug's hidden embedding that is used to determine which drugs it shares an edge with.

- Calculate the Euclidean distance between each drug's hidden embedding. If the distance is below an arbitrary threshold, an edge exists between the drug pair.

  - $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Check $d \leq$ threshold.

This process can be visualized as picking random points in a 2-dimensional plane. Then, if the distance between the two points is sufficiently small, they share an edge. The following could be the 2D visualization of the hidden embeddings of our example graph:

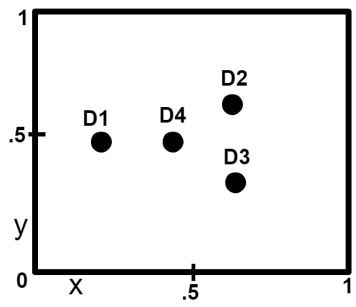

Figure 3.2: Hidden Embeddings of our Example Graph.

While the hidden embeddings are randomly selected, there is a logical and consistent reason for an edge existing between two nodes. Therefore, it is possible to make accurate predictions on the graph.

The hidden embedding of each drug can be thought of as the drug's true feature values. There is only a small number of features, and the relationship between drugs is fairly simple. Many ML problems often appear more complex than they truly are. We may

believe a problem is dependent on many factors when, in reality, many of these factors are either irrelevant or completely dependent on other factors. For example, we may observe that a drug affects blood pressure and also affects blood sugar levels; and we may think these properties are unrelated. However, the drug may affect blood pressure specifically by affecting blood sugar levels. In reality, we should only care about whether the drug affects blood sugar levels; but uncertainty prevents us from observing this. Often, the problem is actually dependent on a small number of factors, modeled in our problem by the small number of true features of a drug. To simulate the uncertainty of observing a problem's complexity, we artificially stretch the number of features to a much higher dimension. This stretched set of features represent the "observable" features of the problem. Thus, the GCN network never sees the hidden embeddings of the drugs; instead, it only sees the "observable" features.

To stretch the hidden embeddings of each drug to a higher dimensional space, we can simply generate a random $(2 \times F)$ matrix, where F is the desired dimensionality. Then, we can take the dot product of a drug's hidden embedding and the stretch matrix to get a $(1 \times F)$ matrix. This way, instead of seeing the two true features of a drug, the network sees what appears to be a much more complicated problem. The following is an example:

If a drug has the following hidden embedding:

$$\begin{pmatrix} .43 & .75 \end{pmatrix}$$

and our stretch matrix is

$$\begin{pmatrix} .42 & .24 & .83 & .74 \\ .19 & .51 & .68 & .06 \end{pmatrix},$$

then their dot product will yield

$$\begin{pmatrix} .32 & .49 & .87 & .36 \end{pmatrix}.$$

This matrix is $(1 \times F)$. We can do this for the rest of the drug embeddings. It is important that we use the same stretch matrix for each drug embedding; otherwise, the problem would be random and unsolvable. Taking the dot product of each hidden drug embedding and the same stretch matrix creates a $(N \times F)$ feature matrix. Thus, we have a complete adjacency matrix and corresponding feature matrix and can now discuss training.

Similarly to Decagon, we will train on subgraphs of the network with the goal of predicting in edges. Unlike Decagon, however, we do not remove edges from the graph and attempt to repredict them back in. Instead, we do the following:

- Select K nodes randomly from the graph. Create the subgraph of the K nodes. This matrix is $(K \times K)$.
- Create the corresponding subfeature matrix. This matrix is $(K \times F)$.
- Select a random pair of nodes. These are the two nodes we want to predict whether an edge exists between. It is important that at most one of these nodes is in our subgraph; otherwise, the problem would be trivial.

Over time, we hope that the network will learn that the problem is less complicated than it appears and begins to make accurate predictions despite not having the true embeddings, much like a real problem.

With the construction of our theoretical data set and discussion of the network's training method complete, we now discuss the model of our GCN network, which can be found at https://github.com/BenjaminDantonio/ResearchGCN.

The GCN takes the following as input:

- A $(K \times K)$ subgraph of randomly selected nodes.

- A $(K \times F)$ subfeature matrix of the randomly selected nodes.
- A pair of nodes to predict whether an edge exists between them, where at most one node is in the subgraph.

And outputs:

- A single value: 0 or 1, where a 1 indicates that the GCN believes the two nodes have an edge and, thus, believes that the two drugs would cause a side-effect.

The following is the GCN's model:

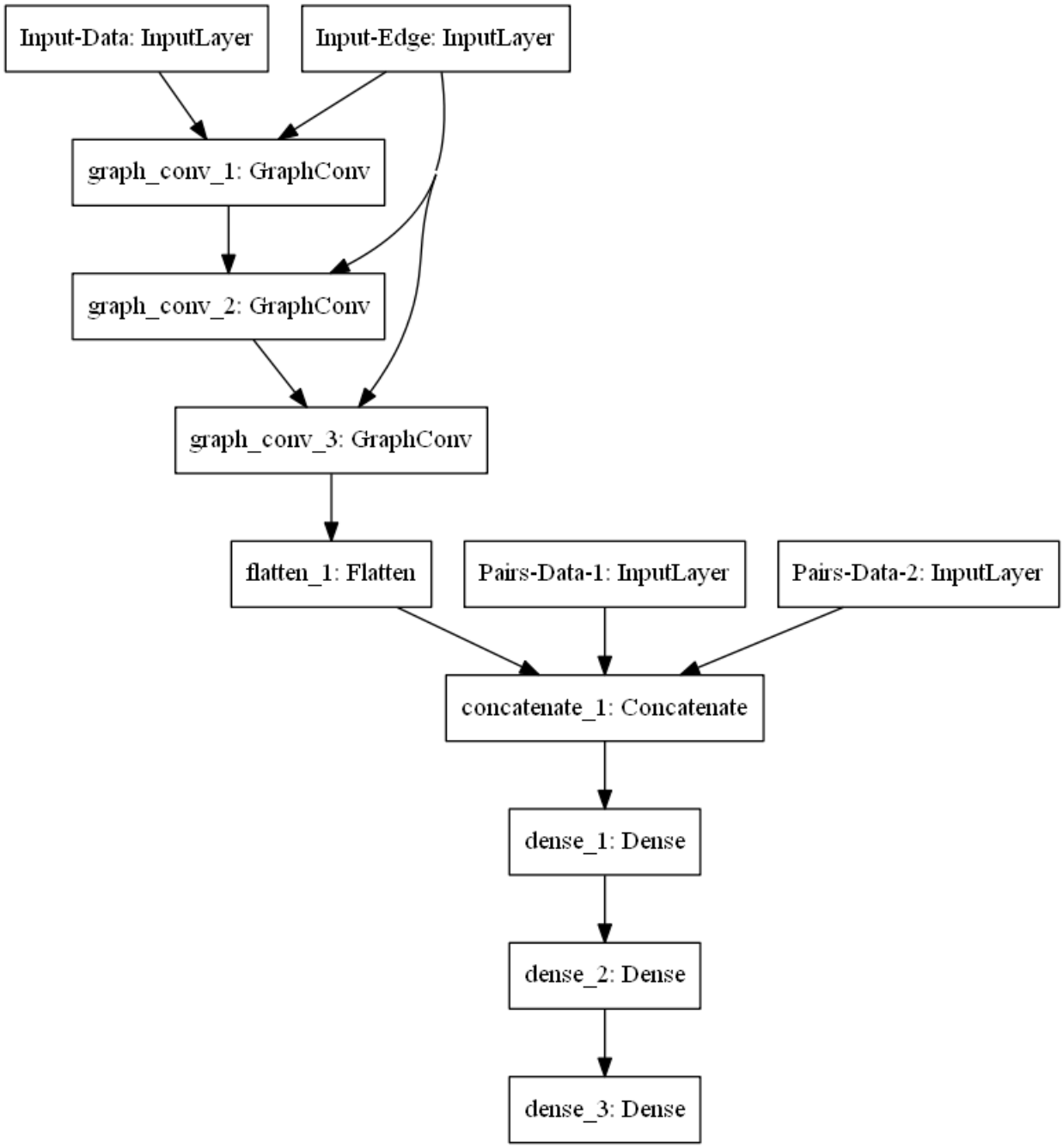

Figure 3.3: Model of our GCN.

`Input-Data` is the subfeature matrix, and `Input-Edge` is the subgraph of the adjacency matrix. These are both passed into the first graph-convolutional layer, `graph_conv_1`. The output of `graph_conv_1` is a convolved feature matrix. This matrix is $(K \times F_1)$, where F_1 is the arbitrary number of output features for `graph_conv_1`. This continues for the next two graph-convolutional layers, generating a $(K \times F_2)$ matrix and finally a $(K \times F_3)$ matrix. $F > F_1 > F_2 > F_3$, so at each step, we convolve the feature matrix into a smaller size. The final feature matrix of the third graph-convolutional layer is sufficiently dense to make a prediction with. We pass this final feature matrix into a flatten layer, which simply "flattens" the matrix into one row. That is, our $(K \times F_3)$ matrix becomes $(1 \times KF_3)$, where each row is concatenated to the first row.

Thereafter, we take the feature vectors of the two drugs we are predicting an edge between (`Pairs-Data-1` and `Pairs-Data-2`) and concatenate them to this flat, convolved matrix using `concatenate_1` to get a $(1 \times (KF_3 + 2F))$ matrix, which is just a vector of length $(KF_3 + 2F)$. This could serve as an "input layer" for a normal DNN. We use it the same way here, passing the vector as input to what is essentially a DNN with three layers.

We pass the input layer through three dense layers, which are the standard layer of a DNN consisting of an arbitrary number of neurons, with each successive dense layer having fewer neurons. The final layer, `dense_3` has one neuron with a sigmoid activation function. Thus, the output of this layer is either a 0 or a 1. This represents the network's prediction for whether an edge exists between the two provided nodes. As with any DNN, the network adjusts its activation values based on feedback on its prediction.

4. EXPERIMENTAL RESULTS

We trained the network and evaluated its performance on the following metrics:

- **Accuracy**: The percentage of correct predictions over total predictions. If the network predicted correctly 9 out of 10 times, it would have a 90% accuracy.

- **Loss**: A measure of how well our network is modeling the dataset. A smaller loss value indicates less uncertainty and, thus, indicates that our network is learning the data. We use binary crossentropy as our loss function, as it is most appropriate for making binary classifications. Binary Crossentropy is calculated as follows:

$$BCE_p = -\frac{1}{N}\sum_{i=1}^{N} y_i * log(p(y_i)) + (1 - y_i) * log(1 - p(y_i))$$

  This formula is not as complicated as it appears. N is the number of predictions being made. Thus, we are taking a summation on our predictions. The summation can be broken into two halves along the $+$. The second half is the compliment of the first. y_i is the actual answer that the prediction should be. Thus, if two drugs cause a side-effect, $y_i = 1$, else 0. $p(y_i)$ is the prediction's probability that two drugs will cause a side-'effect, a value between 0 and 1. Thus, the first half of the summation will, for every time a prediction's answer is 1, add the log of $p(y_i)$ to the total; else it will add 0. The second half of the summation will, for every time a prediction's answer is 0, add the log of the compliment of $p(y_i)$, $1 - p(y_i)$, to the total; else, it will add 0. This way, we calculate the distance between the actual answer and the network's prediction for every prediction, measuring the amount of uncertainty in the network's model of the dataset.

- F_1 **Score**: A measure of accuracy useful in binary classification where the ratio of 1's to 0's is not even. F_1 Score measures how well the network is learning– not predicting. F_1 Score becomes important when the answer is 1, for example, 90% of the time. If the network simply outputs 1 every time, it will have a 90% accuracy. This is deceptive because the network needs no understanding of the data set to achieve 90% accuracy. The F_1 Score is a value between 0 and 1, where 1 is the best possible F_1 Score, and measures the Precision (p) and Recall (r) of a network's predictions to determine if the network is blindly guessing or actually making predictions. F_1 Score is calculated as follows:

$$F_1 = 2\frac{pr}{p+r}$$

Precision is a measure of true positives (tp) and false positives (fp) and can be calculated as follows:

$$p = \frac{tp}{tp+fp}$$

So Precision is true positives over all predicted positives. Thus, if a network outputs 1 every time, it will accrue a lot of true positives but also a lot of false positives. Overall, this will lower the network's Precision.

Recall is a measure of true positives (tp) and false negatives (fn) and can be calculated as follows:

$$r = \frac{tp}{tp+fn}$$

So Recall is true positives over all actual positives. Thus, if a network outputs 1

every time, it will accrue a lot of true positives and no false negatives. Overall, this will increase the network's Recall.

Putting Precision and Recall together, if a network outputs 1 every time and earns a 90% accuracy, its F_1 score will still be low because its Precision and Recall will be balanced/harmonized together, indicating that the network is not actually learning.

4.1 Performance Evaluation

These metrics were gathered for both training and testing phases, so we have Training Accuracy, Testing Accuracy, etc., totaling six metrics. The six metrics were gathered for three tests with differing graph edge densities: 10%, 25%, and 50%. Graph edge density is the measure of the actual number of edges in the graph to the potential number of edges. For example, in a graph with ten drugs, the number of potential edges is $(10^2 - 10)/2$. Because we have ten drugs, the graph can be represented as a (10×10) adjacency matrix– 100 possible slots. Removing the option for self-edges removes 10 slots. The graph is undirected, so if (x, y) in the matrix $= 1$, (y, x) in the matrix also $= 1$, which halves the total number of potential edges. Thus, we arrive at the total number of potential edges being $(10^2 - 10)/2 = 45$ for ten drugs. If 5 of those edges randomly end up being a 1 due to the arbitrary value of the distance threshold, then the graph has an edge density of $5/45$ = 11%. Additionally, for each of the three densities, we tested the following two factors independently and then together: doubling the number of drugs in the network and doubling the number of observable features for each drug. Thus, we performed three edge density tests for the following four trials: a graph with base values (which will be specified), a graph with twice the number of drugs as base, a graph with twice the number of observable features as base, and a graph with both twice the number of drugs and twice the number of observable features as base. With these tests, we demonstrate the effectiveness of our GCN to learn our Polypharmacy problem across different edge densities even as the

problem increases in difficulty.

The following are the base values of the graph which the GCN trained on:

- Number of Drugs: 2000.
- Number of Latent Features: 2.
- Number of Observable Features: 32.
- Number of Subgraphs: 1500. (Not guaranteed to be unique.)
- Number of Drugs in a Subgraph: 10.
- Number of Drug Pairs for Each Subgraph: 8. (Guaranteed to be unique per Subgraph. At most one drug appears in the Subgraph.)

The following are the GCN's training parameters:

- Validation Split: 10%. (The percentage of the Subgraph/Drug Pairs reserved for testing.)
- Number of Epochs: 20.
- Batch Size: 16.

4.2 Trial 1

This trial contains the results of the three edge densities on the base values of the graph. Each metric is measured as its average over all epochs.

Results (Average)			
Metric\Edge Density	10%	25%	50%
Training Accuracy	0.972	0.965	0.959
Training Loss	0.066	0.081	0.095
Training F_1 Score	0.699	0.902	0.955
Test Accuracy	0.977	0.961	0.964
Test Loss	0.0521	0.093	0.083
Test F_1 Score	0.742	0.909	0.958

In each edge density test, the network's performance far exceeded that of being random guesses. Each test achieved over 95% average accuracy for both training and testing. The loss in all cases was very small, as expected with high accuracy. It should be noted that despite a high accuracy on the 10% edge density test, the network's F_1 score took a moderate downturn. This is likely something that can be corrected with proper parameter tuning. However, a lower edge density is generally a harder problem to achieve a high F_1 score on, so the result is not unexpected. Aside from the 10% test, the F_1 score was particularly strong. Overall, the network seems to be fitting very well, as the training and testing results are similar and strong.

4.3 Trial 2

This trial contains the results of the three edge densities on the base values of the graph, except with the number of drugs doubled from 2000 to 4000. Each metric is measured as its average over all epochs.

Results (Average)			
Metric\Edge Density	10%	25%	50%
Training Accuracy	0.972	0.960	0.958
Training Loss	0.068	0.090	0.096
Training F_1 Score	0.656	0.894	0.954
Test Accuracy	0.972	0.962	0.971
Test Loss	0.069	0.083	0.071
Test F_1 Score	0.696	0.904	0.968

This trial's results are similar to Trial 1's. The network's average accuracy never drops below 95%. Similarly, the loss was very small. While the F_1 Score of the 10% test was lower than Trial 1's, the other tests achieved a similar F_1 Score. Even with the increased number of drugs, the network fits strongly. Overall, this indicates that the GCN is capable of scaling well with the size of the graph, as doubling the number of drugs in the graph had little effect on the network's performance.

4.4 Trial 3

This trial contains the results of the three edge densities on the base values of the graph, except with the number of observable features of each drug doubled from 32 to 64. Each metric is measured as its average over all epochs.

Results (Average)			
Metric\Edge Density	10%	25%	50%
Training Accuracy	0.970	0.962	0.957
Training Loss	0.070	0.087	0.096
Training F_1 Score	0.675	0.896	0.955
Test Accuracy	0.979	0.971	0.954
Test Loss	0.051	0.068	0.102
Test F_1 Score	0.717	0.919	0.951

This trial's results hold true to the previous two trials'. From this, we can conclude that increasing the apparent complexity of a drug's features does not hinder the network from performing strongly. That is, it can still understand the underlying, less complex nature of a drug's features despite the presented complexity.

4.5 Trial 4

This trial contains the results of the three edge densities on the base values of the graph, except with the number of drugs doubled from 2000 to 4000 and the number of observable features of each drug doubled from 32 to 64. Thus, Trial 4 is a combination of Trial 2 and 3. This trial questions whether both of these factors scaling together might cause an interaction that negatively affects the network's performance. Each metric is measured as its average over all epochs.

Results (Average)			
Metric\Edge Density	10%	25%	50%
Training Accuracy	0.971	0.961	0.957
Training Loss	0.067	0.087	0.098
Training F_1 Score	0.666	0.897	0.953
Test Accuracy	0.970	0.970	0.957
Test Loss	0.067	0.073	0.101
Test F_1 Score	0.637	0.892	0.956

For the 25% and 50% edge density tests, the simultaneous increase in the number of observable feature values and number of drugs in the graph appeared to have little to no effect. However, in the 10% test, the Test F_1 Score dropped a moderate amount. This likely indicates that the effect of these two features' interaction becomes more pronounced when the network has fewer positive examples to learn from. Whether this can be resolved with proper tuning remains to be investigated. Despite the lower F_1 Score, however, the network still performed as strongly as previous trials in the 10% test.

4.6 Experimental Results Summary

Overall, the GCN performed around the same in terms of accuracy for each test in each trial. The GCN performed slightly worse in terms of loss across all trials as the edge density approached 50%. However, this is likely not significant. The GCN performed better in terms of F_1 Score as the edge density approached 50%. This is notable. It indicates that the network has more difficulty with lower density networks, which is not wholly surprising. Perhaps tuning the network better could improve its performance on low density graphs.

It should be noted that the GCN was created to be as generic as possible. It has three graph convolutional layers and three dense layers to mirror each other, and their parame-

ters' values are simply based on powers of two to stay generic. A great amount of tuning is possible, which would likely increase the effectiveness of the network for any of these trials or edge densities. Despite this, the network still performed strongly.

5. CONCLUSIONS

In this thesis, we have shown that the field of Polypharmacy stands to benefit greatly from ML. We have presented the foundations for ML, DL, and DNNs. We have surveyed current methods of applying ML to Polypharmacy problems, namely GCNs. We have rigorously defined a theoretical Polypharmacy problem and created a GCN that strongly models the problem. This base model demonstrates the potential for GCNs to learn on a Polypharmacy problem.

There is still much work to be done, however. Further steps include the following:

- Tuning the network to better model the theoretical problem.
- Generalizing the theoretical model to fit other models.
- Applying the network to a real-world Polypharmacy problem.

Tuning the network would certainly help the network's overall performance. In particular, tuning the network to have a higher F_1 Score for low density networks is the most important improvement to be made currently. This is because many Polypharmacy problems have low density networks, as drugs do not commonly interact with many other drugs.

Generalizing the theoretical model to be applicable to other models would be a great boon in understanding GCN's potential in Polypharmacy. For example, instead of using a simple, undirected drug graph, the network could learn on a nonsimple, undirected graph where drugs can share multiple edges. Each edge could represent a certain side-effect. Thus, for a given drug pair, the network would predict whether the two drugs' interaction causes each side-effect. Another option is to instead attempt to model a graph with the addition of protein nodes which have their own interactions with drugs and other proteins.

This would provide more information to the network, as two drugs that interact with the same protein may be more likely to cause a side-effect together.

Once the model has been generalized enough, it would be interesting to see how it performs on a real Polypharmacy problem. For example, given a historical database of drug interactions similar to our model, the network could learn to predict whether two drugs that have not been used together will cause a side-effect.

We look forward to future research expanding on the application of GCNs to Polypharmacy problems.

REFERENCES

[1] N. Masnoon, S. Shakib, L. Kalisch-Ellett, and G. Caughey, "What is polypharmacy? a systematic review of definitions," *BMC Geriatrics*, vol. 17, 2017.

[2] G. V. Norman, "Drugs, devices, and the fda: Part 1: An overview of approval processes for drugs," *JACC: Basic to Translational Science*, vol. 1, pp. 170–179, 2016.

[3] K. Quinn and N. Shah, "A dataset quantifying polypharmacy in the united states," *Scientific data*, vol. 4, 2017.

[4] R. Payne and A. Avery, "Polypharmacy: One of the greatest prescribing challenges in general practice," *The British journal of general practice : the journal of the Royal College of General Practitioners*, vol. 61, 2011.

[5] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016.

[6] W. Song, Z. Xiao, Y. Wang, L. Charlin, M. Zhang, and J. Tang, "Session-based social recommendation via dynamic graph attention networks," in *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, WSDM '19, (New York, NY, USA), pp. 555–563, ACM, 2019.

[7] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, (New York, NY, USA), pp. 974–983, ACM, 2018.

[8] M. Zitnik, M. Agrawal, and J. Leskovec, "Modeling polypharmacy side effects with graph convolutional networks," *Bioinformatics*, vol. 34, pp. i457–i466, 2018.

[9] J. Liu, C. Friedman, and J. Finkelstein, "Pharmacogenomic approaches for automated medication risk assessment in people with polypharmacy," *AMIA Joint Summits on Translational Science*, vol. 2017, pp. 142–151, May 2018.

[10] S. Kocbek, P. Kocbek, A. Stozer, T. Zupanic, T. Groza, and G. Stiglic, "Building interpretable models for polypharmacy prediction in older chronic patients based on drug prescription records," *PeerJ*, vol. 6, Oct 2018.

[11] D. Keine, M. Zelek, J. Q. Walker, and M. N. Sabbagh, "Polypharmacy in an elderly population: Enhancing medication management through the use of clinical decision support software platforms," *Neurology and Therapy*, Mar 2019.

[12] H. Gao, Z. Wang, and S. Ji, "Large-scale learnable graph convolutional networks," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, (New York, NY, USA), pp. 1416–1424, ACM, 2018.

[13] W. Song, Z. Xiao, Y. Wang, L. Charlin, M. Zhang, and J. Tang, "Session-based social recommendation via dynamic graph attention networks," in *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, WSDM '19, (New York, NY, USA), pp. 555–563, ACM, 2019.

[14] R. Hu, S. Pan, J. Jiang, and G. Long, "Graph ladder networks for network classification," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, (New York, NY, USA), pp. 2103–2106, ACM, 2017.